Building a Neural Network for Multi-Class Classification

Hikaru Isayama
University of California San Diego
hisayama@ucsd.edu

Avi Mehta
University of California San Diego
avmehta@ucsd.edu

Abstract

This paper presents the development and training of a multi-layered neural network to classify images in the FashionMNIST dataset. We detail the step-by-step process of building, testing, and implementing the neural network from scratch. The dataset used consists of 60,000 training samples and 10,000 test samples, with the training set further split into an 80-20 ratio for validation. All data were normalized and one-hot encoded as part of the preprocessing pipeline. We began by constructing a single-layer neural network with a softmax output activation function. Training was performed using mini-batch stochastic gradient descent with a cross-entropy loss function. This initial single-layer model, incorporating both forward and backward propagation, achieved a test accuracy of around 84.26% and a test loss of around 0.4548 after 100 epochs. Building on this foundation, we introduced hidden layers, momentum, and regularization techniques to enhance the network's performance and robustness. These improvements culminated in a multi-layered neural network that achieved a test accuracy of approximately 88% and a test loss of around 0.33. This progression highlights the effectiveness of iterative development and the impact of advanced optimization techniques on model accuracy and loss reduction.

1 Data Visualization

1.1 Fashion-MNIST

The Fashion-MNIST dataset consists of 28×28 grayscale images of clothing items and accessories, divided into 10 categories. Two example images from the dataset are shown in Figure 1, with their corresponding class labels.

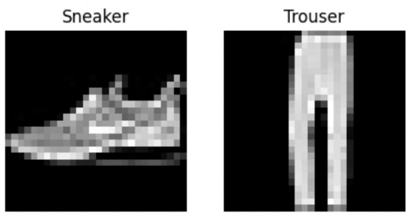


Figure 1: Examples from Fashion MNIST

1.2 Data Loading and Preprocessing

The Fashion-MNIST dataset consists of 60,000 training examples and 10,000 test examples, with each example being a 28×28 grayscale image. To prepare the dataset for training, several preprocessing steps were applied:

Submitted to the Workshop on Instruction Tuning and Instruction Following at NeurIPS 2023. Do not distribute.

1. **Splitting the Dataset:** The training set was divided into 48,000 training examples (80%) and 12,000 validation examples (20%), ensuring a consistent validation set for hyperparameter tuning and performance monitoring.
2. **Normalization:** The pixel values of each image were standardized using z-scoring. For every image, the mean pixel value was subtracted, and the result was divided by the standard deviation. This normalization ensures the pixel values are centered around 0 with a standard deviation of 1, which helps stabilize gradients and accelerates convergence during training.
3. **Label Encoding:** The target class labels were one-hot encoded, transforming each label into a 10-dimensional binary vector where the correct class is represented by a 1, and all other entries are 0.

The mean and standard deviation of a randomly selected image, both before and after normalization, are summarized in Table 1.

Table 1: Summary of Mean and Standard Deviation for an Example Image

Metric	**Before Normalization**	**After Normalization**
Mean	0.28283817	0.28442642
Standard Deviation	-2.6761269e-08	1.00

The dataset was then fed into a multi-layer neural network for classification. The network included an input layer with 784 units, one hidden layer with 128 units, and an output layer with 10 units. Hidden layers used the `tanh` activation function, while the output layer used softmax for generating classification probabilities. Mini-batch stochastic gradient descent was employed for optimization, with cross-entropy loss serving as the objective function.

2 Softmax Regression (Single-Layer Neural Network Implementation)

2.1 Model Initialization

To build up to a neural network for multi-class classification with multiple layers, our team decided it was best to take this step incrementally and first build a neural network with a single layer.

For our first task, we constructed a single-layer neural network where the input layer, consisting of 784 units, is directly connected to the output layer of 10 units using softmax regression, as illustrated in Figure 2. For a given input x_n and c possible categories, the model outputs a vector y_n where each value represents the probability that the input is that respective category. This probability is calculated using the softmax activation function,

$$y_k^n = \frac{\exp(a_k^n)}{\sum_{k'} \exp(a_{k'}^n)} \quad \text{where} \quad a_k^n = w_k^T x^n \tag{1}$$

As shown in Equation 1, the softmax activation function takes the net input a_k^n and transforms the raw scores into probabilities as an output y_k^n. To train the model, we used the cross-entropy cost function for multiple categories (equation 2), which we then took the average of over the number of training examples to normalize the loss over different training set sizes.

$$E = -\sum_n \sum_{k=1}^{C} t_k^n \ln y_k^n \tag{2}$$

Using this loss function, we measure the difference between the true label t_k^n and the predicted probabilities y_k^n outputted by our model. With the softmax activation function and cross-entropy cost function defined, we proceed to initialize the delta rule for this model (as shown in (equation 3).

$$w_{jk} = w_{jk} - \alpha \sum_n \frac{\partial E^n(w)}{\partial w_{jk}} \tag{3}$$

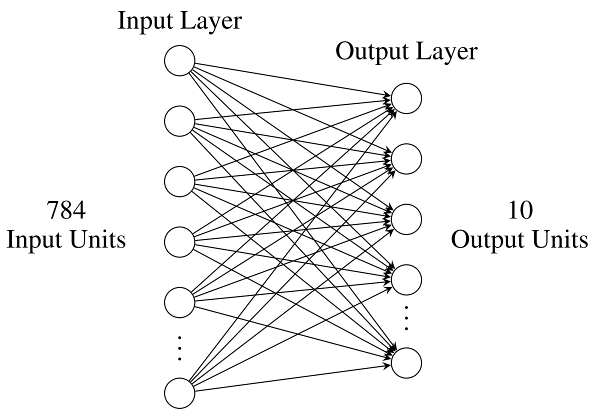


Figure 2: Single-layer neural network with 784 input units and 10 output units.

2.2 Training Procedure

For our training procedure, we employed Stochastic Gradient Descent (SGD) for a single-layer neural network, using mini-batches. In this approach, the weights were initialized with small random values. For each mini-batch, the gradient of the cost function was computed, and the weights were updated using the delta rule (Equation 3). This process was repeated over a specified number of epochs, with the values of each hyperparameter detailed in Table 2.

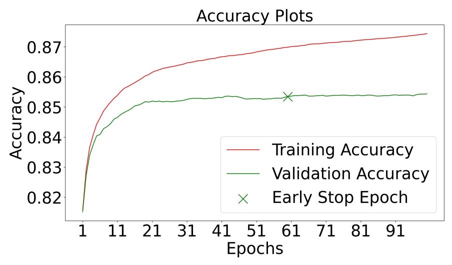
Table 2: Hyperparameters for Stochastic Gradient Descent

Hyperparameter	**Value**	**Description**
Layer specifications	[784, 10]	Number of layers and neurons in each layer
Activation function	None	Type of non-linear activation function
Learning rate	0.01	Step size for gradient descent
Batch size	128	Number of samples per batch
Number of epochs	100	Total number of training iterations
Early stopping	True	Flag to enable early stopping
Early stopping epoch	3	History for early stopping to check validation loss/accuracy
L2 penalty	0	Regularization constant
Momentum	False	Use momentum for training
Momentum gamma	0.9	Value of the 'gamma' parameter in momentum

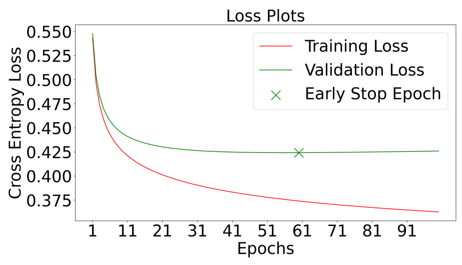
2.3 Results

During training, we measured the accuracy and loss over the set number of epochs to monitor its performance. As shown in Figure 3a, we can observe that the training and validation accuracy appears to increase over epochs, reflecting how the model performance is improving and is able to classify the categories correctly at a higher rate. The initial 20 epochs seemed to result in the greatest improvement in accuracy, and the rest of the epochs improved at a slower rate. A similar result is depicted in Figure 3b, where we can observe that the training and validation loss also appears to decrease over epochs.

At the end of training, the model achieved an accuracy of 84.26% and a loss of 0.4548 on the test dataset, which was in line with the expected performance based on the provided configuration. This result suggests that the model was train correctly and generalizes well on new, unseen data. The loss plots also resemble a exponential decay function, where the loss increased drastically during the first 20 epochs, and starts to stabilize after, suggesting that the model has converged.



(a) Accuracy of Softmax Model



(b) Loss of Softmax Model

Figure 3: Performance metrics for the Softmax model: (a) accuracy and (b) loss.

3 Multi-Layer Neural Network Implementation

3.1 Extending Neural Network Implementation to Multi-Layer

To extend the implementation of our neural network model from single-layer to multi-layer (Figure 4, we modified our code to be able to handle hidden layers and the corresponding forward and backward propagation functions. To ensure the correctness of our new backpropagation implementation, we compared the gradients calculated from backpropagation with values obtained via numerical approximation (Equation 4). Given that the absolute difference between the two gradients are within $O(\epsilon^2)$, this result suggests that the gradients were calculated correctly.

$$\frac{\partial}{\partial w}E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon} \quad \text{where } \epsilon \text{ is a small constant, e.g., } 10^{-2} \tag{4}$$

To check the correctness of our implementation, we chose 5 different weights with an untrained model to compare its gradient and numerical approximation.

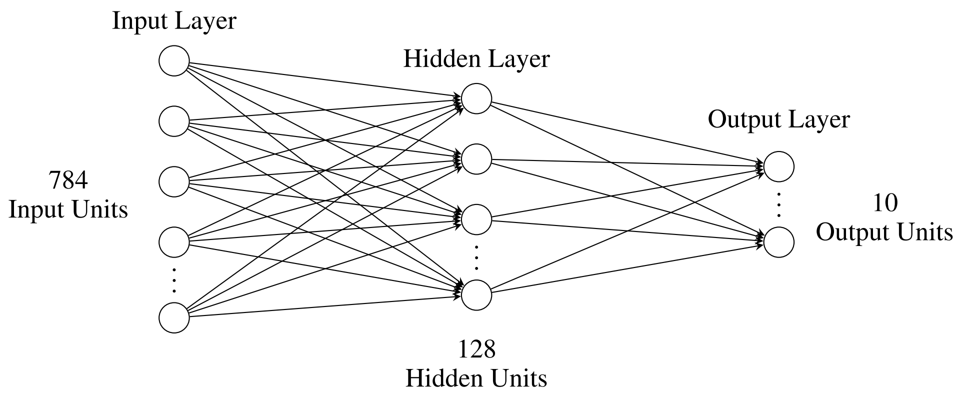


Figure 4: Neural network with ellipses indicating omitted nodes, showcasing decreasing units from input to output.

3.2 Comparing Gradients

Observing the summary of the results of the in Table 3, we can see that the absolute difference is under the $O(\epsilon^2)$ threshold, suggesting that our backpropogation was implemented correctly.

Table 3: Gradients Experiments

Layer	$dw_{i,j}$	Numerical Approximation	Backprop Gradient	Difference
1	[0.0]	0.001004929478010	0.000930350704797	0.000074578773213
0	[0,0]	-0.000176680827457	-0.000164068486977	0.000012612340480
1	[1,0]	-0.002446725245209	-0.002455791114400	0.000009065869191
0	[1,0]	-0.000202470726429	-0.000189530714379	0.000012940012050
1	[2,0]	-0.000258495270211	-0.000292721517504	0.000034226247293
0	[2,0]	-0.000228676244518	-0.000215385035701	0.000013291208817

4 Momentum Experiments

4.1 Training Procedure

For this experiment, we implemented a momentum coefficient to our multi-layer neural network classifier to see if it will improve model performance. Modifying our previous update rule to include the momentum coefficient, our new update rule is as follows (Equation 5).

$$v_t = \gamma v_{t-1} + \alpha \frac{\partial E^n}{\partial w}, \quad w = w - v_t, \text{ where } \gamma = 0.9 \tag{5}$$

Additionally, we implemented early stopping using the validation set – if the validation error went up a certain count above a set "patience" counter in the hyperparameters, the model will stop training and save the weights to store the most optimal validation error. Table 4 consists of the hyperparameters we used for this experiment.

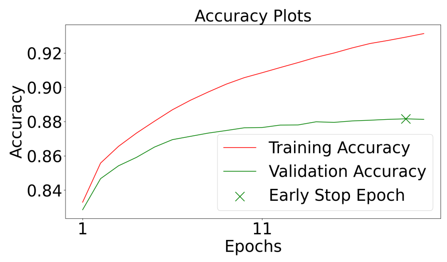
Table 4: Hyperparameter Configuration for Stochastic Gradient Descent with Momentum

Hyperparameter	**Value**	**Description**
Layer specifications	[784, 128, 10]	Number of layers and neurons in each layer
Activation function	tanh	Type of non-linear activation function
Learning rate	0.01	Step size for gradient descent
Batch size	128	Number of samples per batch
Number of epochs	20	Total number of training iterations
Early stopping	True	Flag to enable early stopping
Early stopping epoch	3	History for early stopping to check validation loss/accuracy
L2 penalty	0	Regularization constant
Momentum	True	Use momentum for training
Momentum gamma	0.9	Value of the 'gamma' parameter in momentum

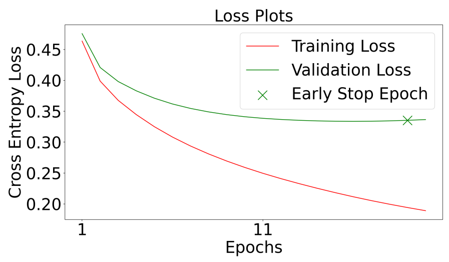
4.2 Results

Using the best weights determined by the new update rule with early stopping, we achieved a test accuracy of 88.36% and loss of 0.3279. Observing the results in Figure 5, we can see that this experiment drastically accelerated the convergence of hour model. Previously, our model converged at around 60 epochs, but now our new model with momentum converged at around 18 epochs with a much higher accuracy. With early stopping, we were able to quickly identify which weights were the most optimal to avoid unnecessary computations and training.

These results demonstrate the importance and significance of using momentum in gradient descent, as we saw that it improved performance by identifying the most optimal weights much faster. By keeping a running average of the previous weights (velocity), the model is able to converge faster, as it more efficiently moves towards the optimal solution



(a) Accuracy of Momentum Model



(b) Loss of Momentum Model

Figure 5: Performance metrics for the Momentum model: (a) accuracy and (b) loss.

5 Regularization Experiments

5.1 Training Procedure

In this experiment, we modified our previous multi-layer neural network by incorporating regularization to improve model generalization. Specifically, we chose to incorporate L1 and L2 regularization into the weight update rule, it will help prevent overfitting by penalizing large weights, helping for a better generalization. L1 regularization adds the absolute values of the weights to the loss function, and L2 regularization will add the squared values of the weights; both techniques will penalize larger weights and help prevent overfitting by reducing model complexity. To implement the two regularization techniques, we modified our backpropagation function to update the gradient by the derivative of the respective regularization factors.

For our training procedure, we kept all hyperparameters the same from our past experiment, but removed momentum, increased the number of epochs by 10% (100 to 110), and added a regularization parameters (λ) of $1e^{-2}$ and $1e^{-4}$. These values for regularization parameters will help us determine the best strength of the regularization; in other words, how strong we want the penalty to be.

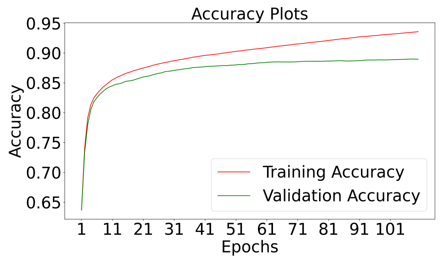
5.2 Results

In the table below (Table 5), we have reported the test accuracy and loss for each of the regularization techniques and regularization parameters. We can observe that the between the L1 and L2 regularization, there does not seem to be a significant difference in the results between the two techniques. The accuracy appears to stays around 88% and the loss appears to stay around 0.33. This suggests that the model is already well-regularized, meaning the model is already generalizing well and not overfitting. This could explain why adding the regularization does not significantly affect the accuracy and loss when compared to the results of our previous experiments.

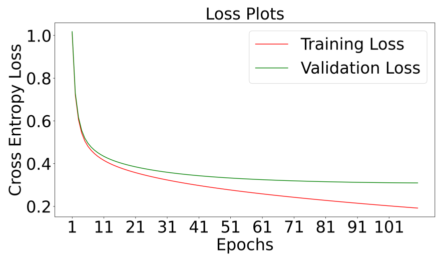
Table 5: Test Accuracy and Loss Metrics

	L1		L2	
λ	1×10^{-2}	1×10^{-4}	1×10^{-2}	1×10^{-4}
Accuracy	0.8825	0.8843	0.8851	0.8819
Loss	0.3426	0.3305	0.3332	0.3322

However, one observation we could make from our plots for accuracy and loss over epoch (Figure 6) for our model using L2 regularization is that the training and validation accuracy appears to be closer together when compared to the plots without regularization. This may suggest that the model is generalizing better, meaning it may be more likely to perform better on new, unseen data.



(a) Accuracy of L2 Model



(b) Loss of L2 Model

Figure 6: Performance metrics for the L2 model using $\lambda = 1e^{-2}$: (a) accuracy and (b) loss.

6 Activation Experiments

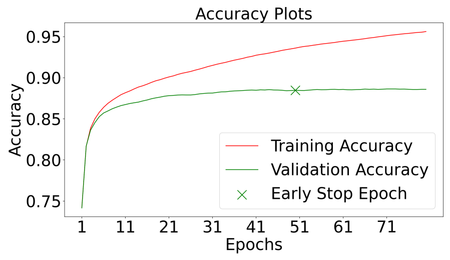
6.1 Testing Alternate Activation Functions

For our last experiment, we explored the performance of different activation functions in a multi-layer neural network. Initially, we used the tanh activation function, but we will now test the Sigmoid and ReLU activation function to observe and compare its results. This comparison is valuable as different activation fucntions can significantly impact the model's ability to learn complex patterns in deep neural networks.

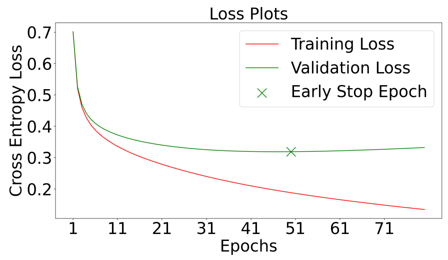
The Sigmoid function is known for producing outputs in the range of 0 to 1, making it useful for binary classification, while ReLU is often preferred for its ability to mitigate the vanishing gradient problem and accelerate convergence in deep networks. By testing these functions, we aim to identify which one allows the model to achieve the best balance between training efficiency and generalization performance.

6.2 Sigmoid Activation Function

Firstly, we experimented using the sigmoid activation function, while keeping all other hyperparameters the same from our previous experiment. Implementing the activation function and calculating the new derivative for the sigmoid, we have plotted the accuracy and loss results in Figure 7.



(a) Accuracy using Sigmoid Activation



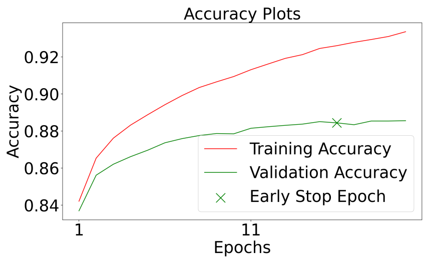
(b) Loss using Sigmoid Activation

Figure 7: Performance metrics using Sigmoid Activation: (a) accuracy and (b) loss.

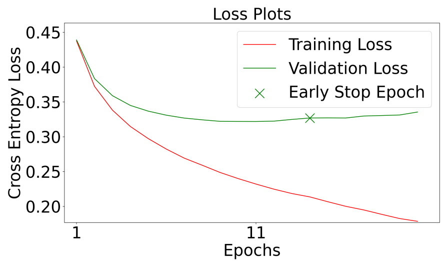
Using the sigmoid activation function in place of the tanh activation function, we achieved a test accuracy of 88.33% and a loss of 0.3441. Comparing these results to our past experiments, we can see that there is not a significanrt increase in performance when using this activation function.

6.3 ReLU Activation Function

Secondly, we experimented using the ReLU activation function, while again keeping all other hyperparameters the same from our previous experiment. Implementing the activation function and calculating the new derivative for the ReLU, we have plotted the train and validation accuracy and loss results in Figure 8.



(a) Accuracy using ReLU Activation



(b) Loss using ReLU Activation

Figure 8: Performance metrics for the ReLU model: (a) accuracy and (b) loss.

Using the ReLU activation function, we achieved a test accuracy of 87.95% and a loss of 0.3649. Comparing the results of using the ReLU activation function in place of the other activation functions we have experimented with, there does not seem to be a significant difference in test performance. However, we can observe a clear difference in the train and validation accuracy and loss plots in Figure 8, where we can see that by using the ReLU activation function, the model appears to converge much quicker than with the other activation functions that we used.

Contributions

Hikaru Isayama = Worked alongside Avi mehta, working on each part together and contributing equally. Went to office hours often to complete all sections.

Avi Mehta = Worked side by side with Hikaru Isayama. Contributed to implementing each step (1-7) in the neural network with Hikaru. Went to office hours quite a bit to get help, especially on backprop method. Overall, worked side by side with Hikaru, so did a little bit of everything.

We confirm that all teammates contributions were equal.